Sequencer

Suresh Podeti

6 min read · Sep 23, 2023

Introduction

There can be millions of events happening per second in a large distributed system. Commenting on a post on Facebook, sharing a Tweet, and posting a picture on Instagram are just a few examples of such events. We need a **mechanism to distinguish these events from each other**. One such mechanism is the assignment of globally unique IDs to each of these events.

A unique ID helps us identify the **flow of an event in the logs** and is **useful for debugging**. Eg: TraceID

Requirements

The requirements for our system are as follows:

- **Uniqueness:** We need to assign unique identifiers to different events for identification purposes.
- **Scalability:** The ID generation system should generate at least a billion unique IDs per day.
- **Availability:** Since multiple events happen even at the level of nanoseconds, our system should generate IDs for all the events that occur.

Solutions

UUID (Universally unique IDs)

ba87e426–54bf-4fda-9b95-a137f2687bde

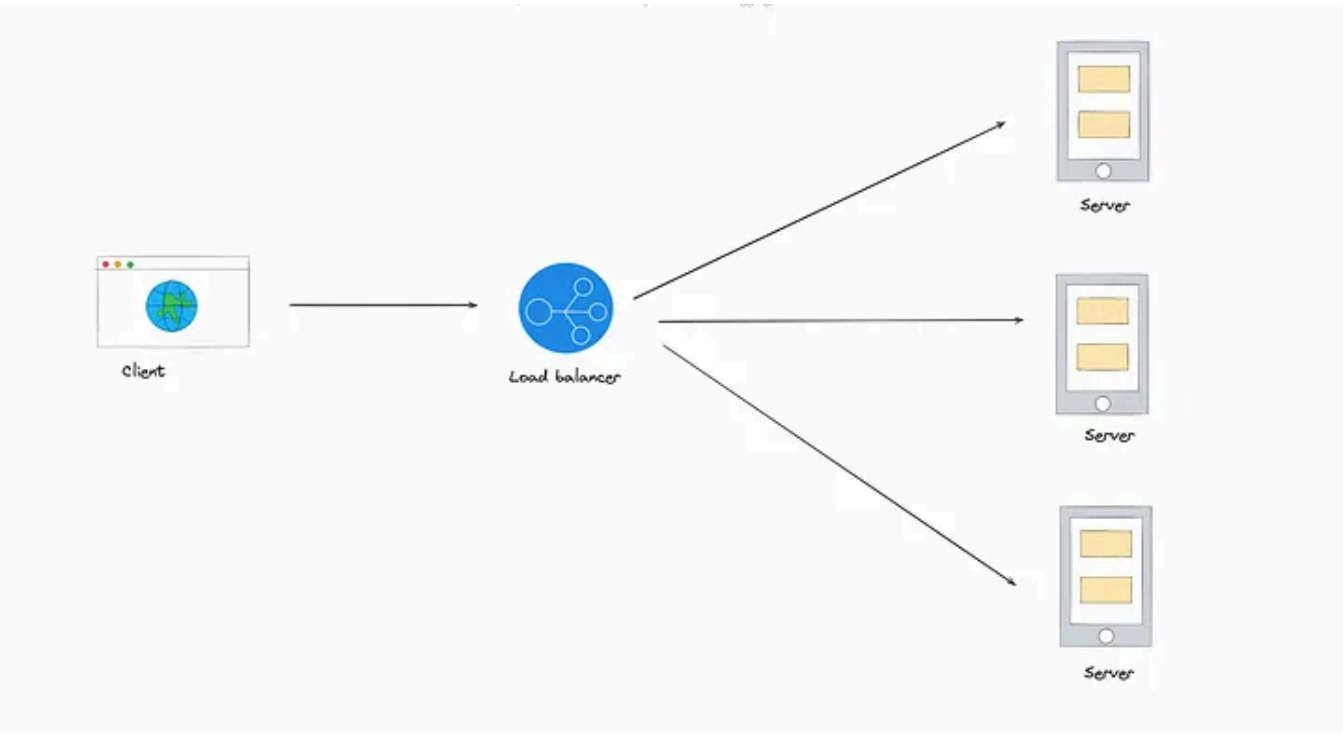

Fig 1.0: Generating a unique ID using the UUID approach

Scaling up and down is easy with UUID, and this system is also highly available. Furthermore, it has a low probability of collisions, and **there's a chance of duplication**

Using a database

Let's try **mimicking** the **auto-increment** feature of a database. Consider a central database that provides a current ID and then increments the value by one. We can use the current ID as a unique identifier for our events.

To cater to the problem of a single point of failure, we modify the conventional auto-increment feature that increments by one. **Instead** of **incrementing by one**, let's rely on a value `m`, where `m` equals the **number of database servers** we have. Each server generates an ID, and the following ID adds `m` to the previous value. This method is **not much scalable.**

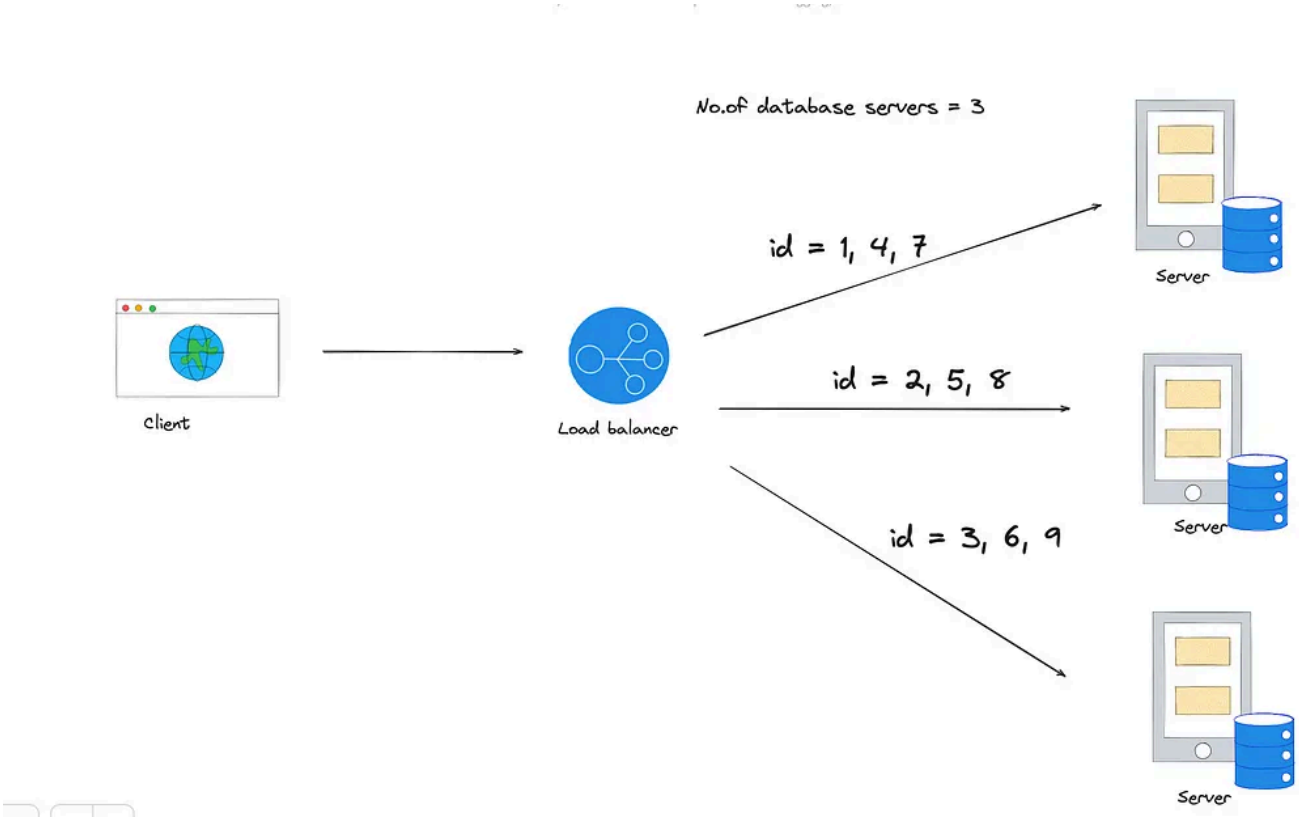

Fig 2.0: Generating IDs using the value of m

It's difficult to scale for multiple data centers. The task of **adding and removing a server can result in duplicate IDs.** For example, suppose `m=3`, and server A generates the unique IDs 1, 4, and 7. Server B generates the IDs 2, 5, and 8, while server C generates the IDs 3, 6, and 9. Server B faces downtime due to some failure. Now, the value `m` is updated to 2. Server A generates 9 as its following unique ID, but this ID has already been generated by server C. Therefore, the IDs aren't unique anymore.

Using a range handler

We can use ranges in a central server. Suppose we have multiple ranges for one to two billion, such as 1 to 1,000,000; 1,000,001 to 2,000,000; and so on. In such a case, a central microservice can provide a range to a server upon request.

Any server can **claim a range** when it needs it for the **first time** or if it **runs out of the range.** Suppose a server has a range, and now it **keeps the start of the range in a local variable.** Whenever a request for an ID is made, it provides the local variable value to the requestor and **increments the value by one.**

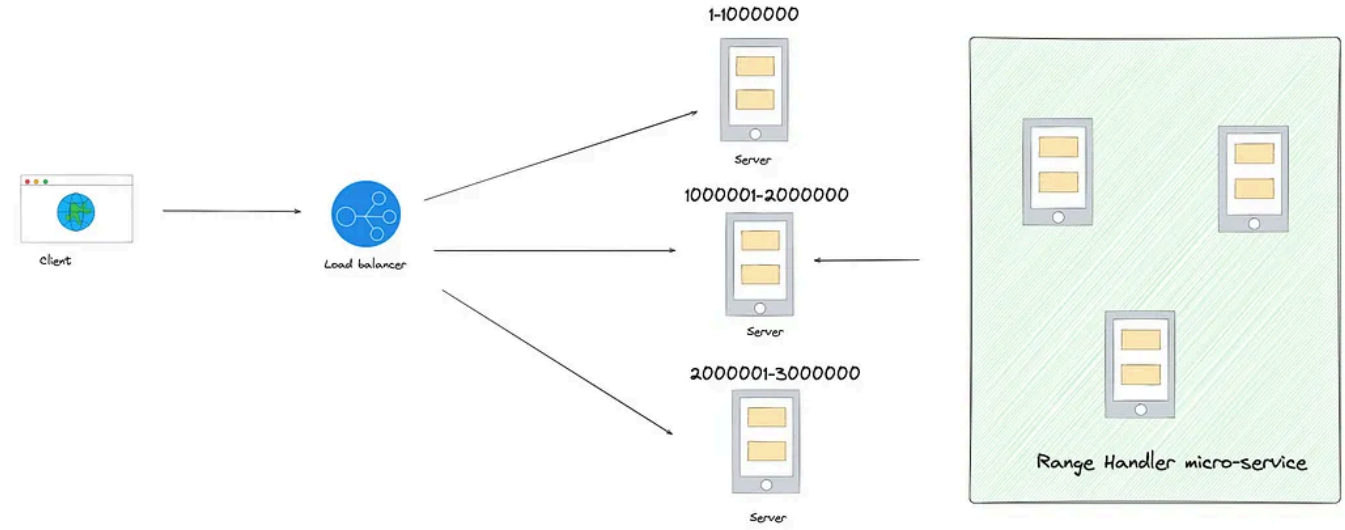

Fig 3.0: Design of the range handler microservice

This resolves the problem of the duplication of user IDs. Each application server can respond to requests concurrently. We can add a load balancer over a set of servers to mitigate the load of requests.

We use a microservice called **range handler** that keeps a record of all the taken and available ranges. The status of each range can determine if a range is available or not. The state — that is, which server has what range assigned to it — can be saved on a replicated storage.

This microservice can become a single point of failure, but a **failover server** acts as the saviour in that case. The failover server hands out ranges when the main server is down. We can recover the state of available and unavailable ranges from the latest checkpoint of the replicated store.

Causality

We have generated unique IDs to differentiate between various events. Apart from having unique identifiers for events, we're also **interested in finding the sequence**

of these events. We need capture the causality i.e **happpend before** relation between events**. A →B ,** event A happend before B**.**

We can not use the timestamp to capture causality as there is no global clock, however, using NTP (Network time protocol) we can synchronise the server timestamps but it will not accurate.

Using logical clocks

We can utilise logical clocks (Lamport and vector clocks) that need **monotonically increasing** identifiers for events

Lamport clocks

In **Lamport clocks**, each node has its counter. All of the system's nodes are equipped with a numeric counter that **begins at zero** when first activated. **Before executing an event, the numeric counter is incremented by one**. The message sent from this event to another node has the counter value. When the other node receives the message, it first updates its logical clock by taking the maximum of its clock value. Then, it takes the one sent in a message and then executes the message.

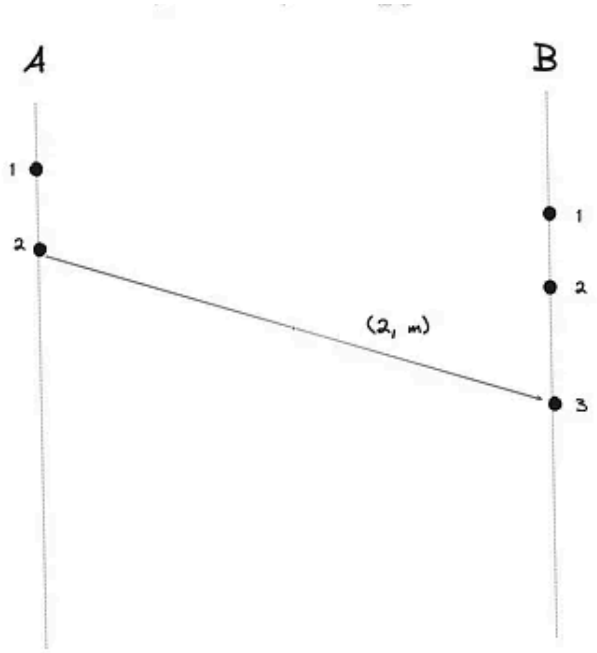

Fig 4.0: Lamport clock

Lamport clocks provide a unique **partial ordering of events** using the happened-before relationship. **We can also get a total ordering of events by tagging unique node/process identifiers**, though such ordering isn't unique and will change with a

different assignment of node identifiers. However, we should note that Lamport clocks don't allow us to infer causality at the global level. This means we can't simply compare two clock values on any server to infer happened-before relationship. Vector clocks overcome this shortcoming.

Vector clocks

Vector clocks maintain causal history — that is, all information about the happened-before relationships of events. So, we must choose an efficient data structure to capture the causal history of each event.

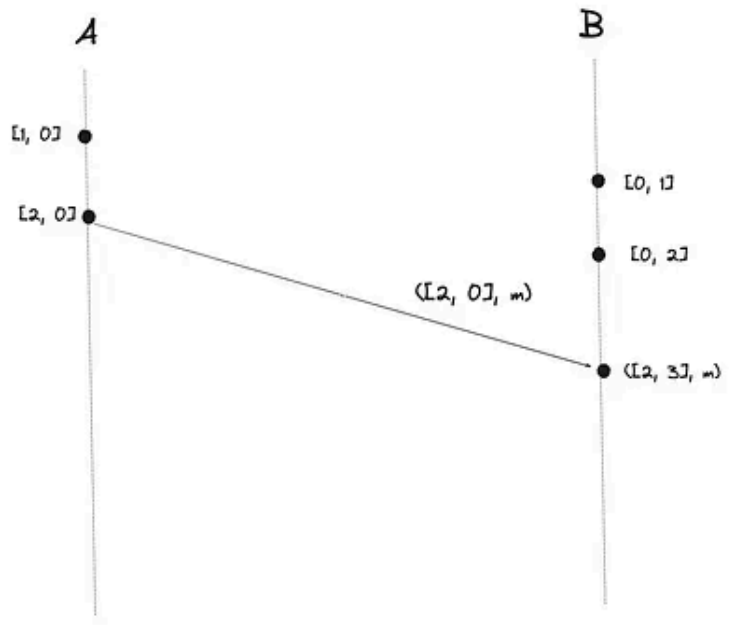

Fig 5.0: Vector clock

Our approach with vector clocks works. However, in order to completely capture causality, a vector clock must be at least *n* nodes in size. As a result, when the total number of participating nodes is enormous, vector clocks require a significant amount of storage. Some systems nowadays, such as web applications, treat every browser as a client of the system. Such information increases the ID length significantly, making it difficult to handle, store, use, and scale.

TrueTime API

Google's TrueTime API in Spanner is an interesting option. Instead of a particular time stamp, it reports an interval of time. When asking for the current time, we get back two values: the **earliest** and **latest** ones. These are the earliest possible and latest possible time stamps.

Based on its uncertainty calculations, the clock knows that the actual current time is somewhere within that interval. The width of the interval depends, among other things, on how long it has been since the local quartz clock was last synchronised with a more accurate clock source.

Google deploys a GPS receiver or atomic clock in each data center, and clocks are synchronized within about 7 ms. This allows Spanner to keep the clock uncertainty to a minimum. The uncertainty of the interval is represented as epsilon.

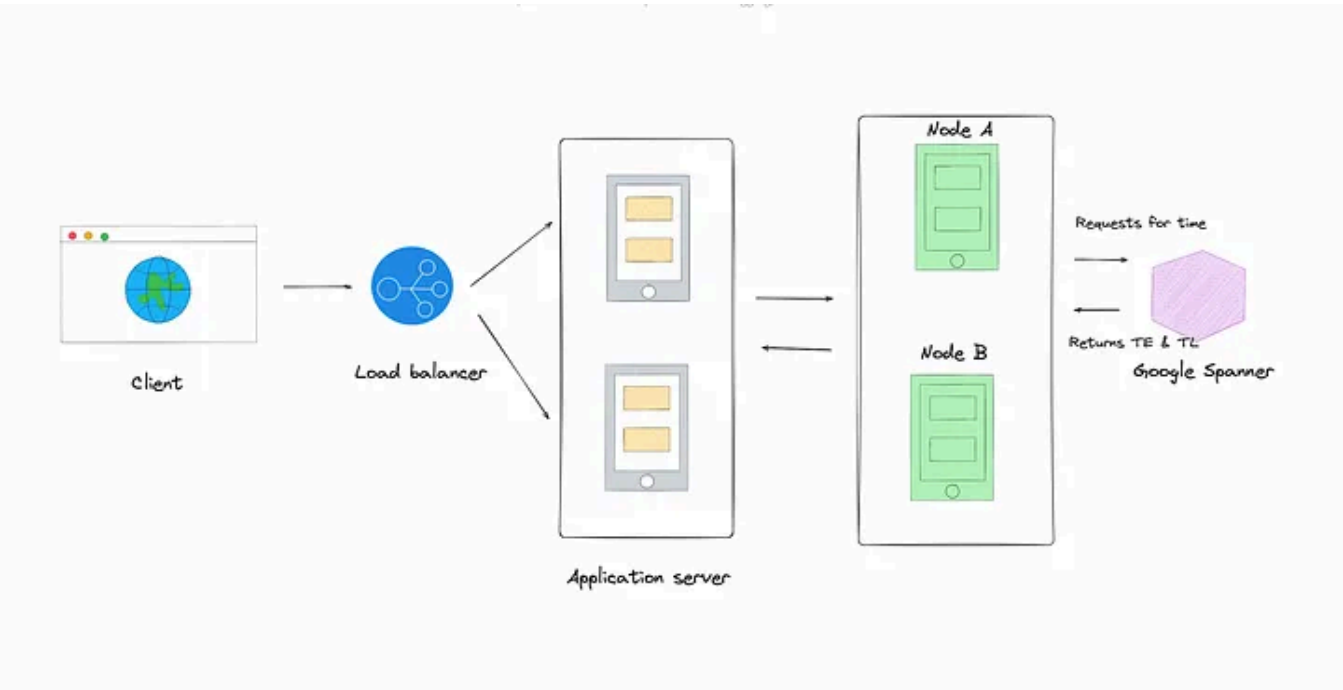

Fig 6.0: Node B generating a unique ID for its event using TrueTime

Sequencer | System Design Interview | Distributed Systems | Software Architect

Edit profile

Written by Suresh Podeti

1.2K Followers